CPV: Open Source Software Survivability Analysis by Communication Pattern Validation Approach

Angel Mary[1], N .Sambasiva Rao[2]

[1]Department of CSE, Vardhaman College of Engineering, Shamshabad, A.P, India
mrs.angelrex@gmail.com

[2]Principal, Vardhaman College of Engineering, Shamshabad, A.P, India
snandam@gmail.com

Abstract: ***The popularity of open source is unassailable. It has been widely adopted for different purposes, including web servers, e-mail servers, operating systems, and programming languages. Although some open source projects, like Apache, Sendmail and Linux, have achieved extraordinary success, lots of projects are lackluster or with no developing activity at all. To deepen our understanding of open source projects, it is essential to explore the factors that have contributed to its success or failure. The objective of this research is to analyze the survival of open source projects from the social network perspective. In particular, we have studied the impact of the communication pattern of open source projects on the project success, defined in view of both the supply side (developers) and the demand side (end users), while incorporating control factors such as project-specific characteristics. Based on empirical data collected from SourceForge.net, a popular open source hosting web site, our results show that the communication pattern has significant impacts on the open source success.***

Keywords: OSS, Communication pattern, OSS developer communities

1. Introduction

Open Source is the term used to describe software, of which the high-level code is supplied free of charge to whoever would like it, by definition, they are permitted to:

- Use the code for whatever they wish
- Edit and modify the code to suit their needs
- Redistribute the code to whoever they wish
- Improve the software, and redistribute it in its modified form
- When all these conditions are met, software is considered truly Open Source.

Open source has played a fundamental role in the development of the Internet by contributing to such remarkable software as TCP/IP, BIND, Sendmail, Linux, and the Apache WEB server. The first works in this rapidly developing field were descriptive in nature followed by theory driven explanations and early empirical research. Many of the early explorations into the inner workings of the open source development process have sought to explain the mechanisms by which open source projects attract and motivate volunteers to produce such seemingly high quality. One aspect, however, of the OSS phenomenon that has received relatively little attention is the nature of the project communication in open source projects. We are specifically interested in advancing the understanding of project communication and its role in managing the process of creating open source software. How open source developers communicate and interact is an interesting and important question given the geographic distribution of the developers and the unstructured process of software development in the open source context (compared to software development in a closed source setting). This study utilizes archival data to explore the nature of the social network and the patterns of communication that exist in an OSS project.

Lerner and Tirole (2005b) suggest four questions/issues of interest to scholars studying open source software: 1) technological characteristics conducive to smooth open source development, 2) optimal licensing of open source, 3) the coexistence of open source and proprietary software, and 4) the potential for the open source model to be carried over to other industries (i.e. the portability of the "open source" concept). While technological issues are useful to the adoption of open source, just as important are social issues. In his examination of the Linux operating system, Weber (2000) identifies three key issues for social scientists to ponder: 1) motivation of individuals who develop open source; 2) coordination of activities in the supposed absence of a hierarchical structure, and 3) growing complexity in open source projects and its management. This paper presents the preliminary findings from a literature review focusing on communication pattern involved in Open Source Software Development.

2. Literature Review

2.1 Software Quality Models: Purposes, Usage Scenarios and Requirements

There is a huge amount of work on various forms of quality models. However, comprehensive overviews and classifications are scarce. A first, broad classification of what he called "quality evaluation models" was proposed by Tian. He distinguishes between the specificness levels generalized and product-specific. These classes are further partitioned along unclear dimensions. For example, he distinguishes segmented models for different industry segments from dynamic models that provide quality trends. Two of the authors built on Tian's work and introduced further dimension. Wagner discussed in the dimensions purpose, quality view, specificness and measurement where the purposes are construction, assessment and prediction. This was further extended by Wagner and Deissenboeck with the dimensions phase and

technique. A thorough discussion of critique, usage scenarios and requirements along these dimensions is not provided in any of these contributions.

An impressive development of quality models has taken place over the last decades. These efforts have resulted in many achievements in research and practice. As an example, take a look at the field of software reliability engineering that performed a wide as well as deep investigation of reliability growth models. In some contexts these models are applied successfully in practice. The developments in quality definition models even led to the standardization in ISO 9126 that is well known and serves as the basis for many quality management approaches.

However, the whole field of software quality models is diverse and fuzzy. There are large differences between many models that are called “quality models”. Moreover, despite the achievements made, there are still open problems, especially in the adoption in practice. Because of this, current quality models are subject to a variety of points of criticism that have to be acted on.

They provided a comprehensive definition of a quality model based on the purpose the model has. Using this tripartion in definition models, assessment models and prediction models (DAP), they summarized the existing critique and collected a unique collection of usage scenarios of quality models. From this, they derived a comprehensive set of requirements, again ordered in terms of the DAP classification, that can be used in two contexts: (1) evaluation of existing models in a specific context or (2) further developments and improvements of software quality models.

2.2 Cave or Community?

An Empirical Examination of 100 mature Open Source Projects:

Starting with Eric Raymond's ground-breaking work, "The Cathedral and the Bazaar”, open-source software (OSS) has commonly been regarded as work produced by a community of developers. Ghosh's cooking pot markets, similarly, point to a communal product development system. Certainly, this is a good label for some OSS products that have been featured prominently in the news. For instance, Moon and Sproull point out that by July 2000, about 350 contributors to LINUX were acknowledged in a credit list in the source code of the kernel.

However, the goal in Eric’s paper is to ask if the community-based model of product development holds as a general descriptor of the average OSS product. He systematically looks at the actual number of developers involved in the production of one hundred mature OSS products. What Eric found is more consistent with the lone developer (or cave) model of production rather than a community model (with a few glaring exceptions, of course).

This is not to say that there is no community in the OSS movement. For instance, the findings of Butler, Kiesler, Sproull and Kraut (2002) point to participation by individuals other than the creators of OSS-program-related mailing lists. Their contention is only that communities do things other than produce the actual product - e.g. provide feature suggestions, try products out as lead users, answer questions etc. Formally separating software production from other steps in the development of OSS programs will provide greater clarity to the discussion of the OSS phenomenon.

2.3 Extracting Facts from Open Source Software

The open source software is usually developed outside companies – mostly by volunteers – the quality and reliability of the code may be uncertain. Thus its use may involve big risks for a company that uses open source code. It is vital to get more information out of these systems. Various kinds of code measurements can be quite helpful in obtaining information about the quality and fault-proneness of the code. These measurements can be done with the help of proper tools.

The paper describes a framework called Columbus with which we are able to calculate the object oriented metrics validated for fault-proneness detection from the source code of the well-known open source web and e-mail suite called Mozilla. We then compare our results with those presented. One of their aims was to supplement their work with metrics obtained from a real-size software system. They also compare the metrics of the seven most recent versions of Mozilla (1.0–1.6), which covers over one and a half years of development, to see how the predicted fault-proneness of the software system changed during its development.

The Columbus framework has been further improved recently with a compiler wrapping technology that allows us to automatically analyze and extract information from practically any software system that compiles with GCC on the GNU/Linux platform (the idea is applicable as well to other compilers and operating systems).

Rudolf Ferenc et.al also introduce a fact extraction process to show what logic drives the various tools of the Columbus framework and what steps need to be taken to obtain the desired facts. One fact might be, for instance, the size of the code. Another fact might be whether a class has base classes. Actually any information that helps us better understand unknown source code is called a fact here. It is obvious that collecting facts by hand is only feasible when relatively small source codes are investigated. Real-world systems that contain several million lines of source code (Mozilla, for instance) can only be processed with the help of tools.

In the Columbus framework the form of the extracted facts conform to predefined schemas. By schema, we mean a description of the form of the data in terms of a set of entities with attributes and relationships. A schema instance is an embodiment of the schema which models a concrete software system (or part of it). This concept is analogous to databases, which also have a schema

(described usually by E-R diagrams) that is distinct from the concrete instance data (data records). The Columbus framework defines two schemas: (1) the Columbus Schema for C/C++ Preprocessing [15] (for describing preprocessing related facts) and (2) the Columbus Schema for C++ [7] (for describing the C++language itself).It should be mentioned here that we performed full analyses of the seven versions of Mozilla and built up the full schema instances of them, which can be used for any re and reverse engineering task like architecture recovery and visualization. In this work we used them only for calculating metrics, but their use is not limited to this case.

Mozilla was investigated earlier by Godfrey and Lee. They examined the software architecture model of Mozilla Milestone-9. The authors used the PBS and Acacia reverse engineering systems for fact extraction and visualization. They created the subsystem hierarchy of Mozilla and looked at the relationships among them. Their model consists of 11 top-level systems which may be divided into smaller subsystems. They created the subsystem hierarchy by taking into consideration things like source directory structure and software documentation. It turned out that the dependency graph was near complete, which means that almost all the top-level systems use each other.

Fioravanti and Nesi took the results of the same projects as Basili et al. to examine how metrics could be used for fault-proneness detection. They calculated 226 metrics and their aim was to choose a minimal number relevant for obtaining a good identification of faulty classes in medium-sized projects. First they reduced the number of metrics to 42 and attained very high accuracy score (more than 97%). This model was still too large to be useful in practice. By using statistical techniques based on logistic regression they created a hybrid model which consists of only12 metrics with an accuracy that is still good enough to be useful (close to 85%). The metrics suite they obtained is not the same as the one used in [1] but there are many similarities.

Yu, Syst¨a and M¨uller chose eight metrics in [16] (actually ten because CBO and RFC was divided into two different kinds) and they examined the relationship between these metrics and the fault-proneness. The subject system was the client side of a large network service management system developed by three professional software engineers. It was written in Java and consisted of 123 classes and around 34, 000 lines of code. First they examined the correlation among the metrics and found four highly correlated subsets. Then they used univariate analysis to find out which metrics could detect faults and which could not. They found that three of the metrics (CBO in, CBO out and DIT) were unimportant while the others were significant but to different extents (NMC, LOC, CBO out, RFC out, LCOM, NOC and Fan-in).

The paper makes three key contributions: (1) we presented a method and toolset with which facts can be automatically extracted from real-size software; (2) using the collected facts we calculated object oriented metrics and supplemented a previous work [1] with measurements made on the real-world software Mozilla; and (3) using the calculated metrics we studied how Mozilla's predicted fault proneness has changed over seven versions covering one and a half years of development.

In the future they planned to validate the object oriented metrics and hypotheses presented (and here as well) on Mozilla for fault-proneness detection using the reported faults which are available from the Bugzilla database. We also plan to scan Mozilla (and also other open source systems) regularly for fault-proneness and make these results publicly available.

2.4 Assessing the Health of Open Source Communities

The computing world lauds many Free/Libre and Open Source Software offerings for both their reliability and features. Successful projects such as the Apache http Web server and Linux operating system kernel have made FLOSS a viable option for many commercial organizations.

While FLOSS code is easy to access, understanding the communities that build and support the software can be difficult. Despite accusations from threatened proprietary vendors, few continue to believe that open source programmers are all amateur teenaged hackers working alone in their bedrooms. But neither are they all part of robust, well-known communities like those behind Apache and Linux.

If you, as an IT professional, are going to rely on or recommend FLOSS, or contribute yourself, you should first research the community of developers, leaders, and active users behind the software to decide whether it's healthy and suitable for your needs.

2.5 Life Cycle and Motivations

Understanding a project's life cycle and its participants' motivations is useful for understanding why a FLOSS community is important to a project's success.

Eric Raymond claims that successful open source projects usually start in a "cathedral" before heading into the "bazaar" ("The Cathedral and the Bazaar," First Monday, vol. 3, no. 3, 1998; www.firstmonday.org/issues/issue3_3/raymond/index.html). AnthonySenyard and Martin Michlmeyr,a former Debian project leader, agree, arguing that a working code base for a successful FLOSS project is usually developed alone or by a very small group before going public ("How to Have a Successful Free Software Project," Proc. 11th Asia-Pacific Software Eng. Conf., IEEE CS Press,2004, pp. 84-91).

The founders' good ideas expressed in working code facilitate a successful project's second phase: a "creative explosion" in which the product, now public, develops quickly, gathering features and capabilities that in turn attract additional developers and users. What Perl founder Larry Wall calls "learning in public" can be an exhilarating, if difficult, time early in project's life cycle.

More broad-based research by Rishab A. Ghosh and colleagues("Free/Libre and Open Source Software: Survey and Study," summary report, Workshop on Advancing the Research Agenda on Free/Open Source Software, Int'l Institute of Infonomics,Univ. of Maastricht, 2002, www.infonomics.nl/FLOSS/report/workshopreport.htm) and Karim Lakhani and Robert G. Wolf ("Why Hackers Do What They Do: Understanding Motivation Efforts in Free/Open Source Software Projects," working paper4425-03, MIT Sloan School of Management,2003, http://opensource.mit.edu/papers/lakhaniwolf.pdf) indicates that motivations are quite diverse and include, in decreasing order of relevance,

• Intellectual engagement;
• Knowledge sharing;
• The product itself; and
• Ideology, reputation, and community obligation.

Projects that have an atmosphere of exploration and intellectual engagement, especially early in their life, are most likely to attract the active user community needed for future success. Also important in attracting good developers is a code base that solves a real need.

Few if any FLOSS projects are likely to apply for ISO 9000 process certification anytime soon, but that doesn't mean they don't have well-accepted processes. However, these are rarely formally documented, and understanding what Walt Sccachi calls "informalisms"can take time ("Understanding the Requirements for Developing Open Source Software Systems," IEE Proceedings—Software, vol. 149, no. 1, 2002, pp. 24-39).

Even very successful open source projects often lack detailed roadmaps, explicit work assignments, or feature request prioritizations. A key aim of making proprietary software processes explicit is to ensure the efficient use of a fixed pool of resources, but FLOSS projects don't face such fixed pools, either in the number of participants or in the amount of time each one can devote.

Therefore, organizing for fun can be more important than organizing for efficiency. In fact, duplication of effort could be a positive sign that the project can attract resources and is in a position to choose the best contributions. On the other hand, a formalized system for prioritizing security issues clearly benefits some applications. And if the processes are discussed, it's important that these discussions regularly end in action that lets people get back to work rather than in an exhausted stalemate.

Certain essential processes, such as providing and maintaining repositories and download sites, are often both difficult and laborious; these have a high burn-out rate because many participants are driven by intellectual curiosity rather than a service mentality. Such functions should be farmed out to an entity with financial and operational resources, or at least formally rotated among participants. Hosting providers such as Source-Forge, Savannah, GForge, and Ruby-Forge do the FLOSS ecosystem a great service in this regard, as do long-term partnerships with commercial ventures.

Managing releases is another onerous task. Hassling volunteers and collaborators to stop being creative and focus on delivering and testing are leasable version isn't particularly enjoyable. Ideally, a healthy FLOSS community recognizes and explicitly addresses this issue, perhaps through a rotating position (like Perl's pump king) or a time-based release strategy.

In considering a FLOSS project, make it a point to understand the community as you familiarizes yourself with the code. Subscribe to and skim mailing lists, find the list archives, and examine the project's Web site. If the project has an Internet relay chat channel, spend some time there—IRC's informality can reveal whether participants actually like one another. Many quantitative resources can reveal how a community has evolved over time. For example, FLOSS mole (http://ossmole.sourceforge.net) provides public access to data collected from Source Forge, freshmeat, and RubyForge, as well as tools for graphical analysis of community structure. CVS AnalY (http://cvsanaly.tigris.org)offers utilities and data extracted from many CVS and Subversion projects, such as the number of contributors and their rate of contribution. The Business Readiness Rating project (http://openbrr.org) is developing a more formal methodology to assess FLOSS projects.

If your assessment leaves you feeling that the community isn't right for you, be prepared to consider alternatives, no matter how attractive the code is. Trying to change an existing community is likely to end in frustration and undermine the reasons you chose FLOSS in the first place. However, while a rejection of your enthusiastic contributions can seem dictatorial and rude, it can also demonstrate a long-term, cohesive vision—a FLOSS community at its best.

3. Methodology

Analyzing mailing lists used by a community of software developers provides insights in how developers work and what are the most important persons involved in the project. Open-source projects typically have a mailing list that channels the communication in the project. Mailman is one of the most used infrastructures for handling mailing lists.

We assess the activity in the mailing list by the number of e-mails that are sent. In a version repository, an activity is basically a commit. By analyzing the evolution of the activity we know if the project is growing, stable or even abandoned. The level of activity is a good indicator of the health of a project.

4. Conclusion

Open source represents an exciting opportunity for research in a wide variety of disciplines. This paper applies social network analysis to understand how developers communicate in an open source project. Since

the developers in open source projects are geographically distributed and may never meet face-to-face, it is important to understand how they communicate to organize and coordinate their efforts. Our future work is to develop a tool to analyze the communication between the open source software developer communities.

References

[1] OSI, the Open Source Definition, The Open Source Initiative, (Accessed: May 2001); http://opensource.org/docs/definition _ Plain.html

[2] R. Abreu and R. Premraj. How developer communication frequency relates to bug introducing changes. In Proc. Joint Int'l Workshop on Software Evolution (IWPSE-EVOL), pages 153-157. ACM

[3] SIGSOFT, 2009.

[4] E. Raymond, The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary (O'Reilly, Cambridge, 1999).

[5] K. Lakhami, and E. von Hippel (2000). How Open Source Software works: "Free" user-to-user assistance. MIT Sloan Open Source Project, (Accessed: October 2001); http://opensource.mit.edu/papers/lakhanivonhippelusersupport.pdf

[6] A. Mockus, R. Fielding and J. Herbsleb, A Case Study of Open Source Software Development: The Apache Server, Proceedings of the Proceedings of the 22nd International Conference on on Software Engineering, Limerick Ireland (2000).

[7] G. Madey, V. Freeh, and R. Tynan. The open source software development phenomenon: An analysis based on social network theory. In Eight Americas Conf.

[8] Information Systems, pages 1806{1813, 2002.

[9] S. Y. T. Lee, et Al, "Measuring open source software success," Omega-International Journal of Management Science, vol. 37, pp. 426-438, 2009

[10] J. Wang, “Survival factors for Free Open Source Software projects: A multi-stage perspective”, European Management Journal, Vol.30(4), pp.352-371, 2012.

[11] W. H. DeLone, E. R. McLean, "The DeLone and McLean model of information systems success: a ten-year update," Journal of Management Information Systems, vol. 19, pp. 9-30, 2003

[12] V. Midha, P. Palvia, “Factors affecting the success of Open Source Software”, Journal of Systems and Software, vol. 85, pp. 895-905, 2012